Duplicating and Verifying LogTM with OS Support in the M5 Simulator

Geoffrey Blake, Trevor Mudge
{blakeg,tnm}@eecs.umich.edu
Advanced Computer Architecture Lab
Department of Electrical Engineering and Computer Science
2260 Hayward Ave
Ann Arbor, MI 48109-2121

ABSTRACT

LogTM is a transactional memory (TM) implementation that is very promising. It requires moderate augmentation of existing hardware and uses software handlers to support complex operations such as rolling back the transactional memory state. LogTM has shown it offers good performance, usually outperforming lock based code while improving programmability.

Currently, research in LogTM and TM in general make simplifying assumptions about how a real system will behave by abstracting away details. One such simplification is to abstract away the OS. In this paper we duplicate LogTM in the M5 simulator which models operating system interactions with hardware rigorously, which disallows us to abstract the OS away from the TM system. We find the OS needs intimate knowledge of the TM extensions to properly share the system with transactional and non-transactional threads. Because we do not abstract away the OS in M5, the OS is able to cause interference in the TM system by aborting transactions. We quantify this OS interference, and find it to be benchmark dependent. The OS causes very few aborts the majority of the time but can cause up to 42% of all aborts. In this paper we also investigate the published performance results from LogTM and find them to be correct when the transactions are assumed to be small. We extend the results using new benchmarks that do not assume small transactions, and find our new benchmarks expose pathological behavior in LogTM with up to 98% of the transactions aborting.

1. INTRODUCTION

Recent trends in microprocessor design have moved from implementing longer and wider pipelines to adding extra cores in order to achieve additional performance. Multi-core processors are now becoming very common in all segments of the industry from embedded devices to the desktop[1], meaning programmers must now concentrate on writing parallel code to extract performance.

Writing parallel code can be difficult with the tools currently available to programmers, i.e., locks and semaphores[8]. Locks and semaphores are difficult to use because the programmer must have very intimate knowledge of the program being written to properly protect shared data structures with the correct mutexes. Programmers also have to follow the proper locking order to prevent deadlock. To ease the job of the programmer, Herihly and Moss[13] proposed transactional memory, a technique that commits groups of memory accesses as one atomic unit. Transactional memory systems take the responsibility of checking transactions against each other to find true dependencies, relieving the programmer from having to explicitly find and protect dependencies with the correct combination of locks. Because of the benefits offered by TM, significant amounts of research have been published recently with particular emphasis on hardware transactional memories (HTMs). HTMs are interesting because they offer the most flexible, general and lowest overhead form of TM and so it has been pursued heavily in many directions. Examples include Stanford's TCC[10], Wisconsin's LogTM[17] and Illinois' Bulk[7]. These competing ideas have shown that TM is a viable solution to providing critical section atomicity and isolation as well as providing better performance.

Currently, little work has been pursued in deconstructing exactly the OS requirements for a user level HTM implementation. Many systems abstract away the OS and concentrate on developing more complex semantics for their TM system, for example, complex nesting. Current work abstracts away the OS in the following ways: Bulk[7] runs on a simulator that only supports user code, and does model the OS. TCC[10] relies on trace based simulations and also does not model the OS. LogTM[17] uses a simulator environment that runs OS code as well as user code, but this implementation makes LogTM visible only to the underlying memory model to avoid having to modify the OS. There has also been little work in the way of benchmarks that would be representative of future applications that will use TM. Currently the benchmarks used for study are small and are primarily database or scientific workloads optimized for older SMP systems. These benchmarks have been shown to work very well with TM because of very small transactions, leading to low conflict rates, but this may not be the case with future applications.

WDDD'07 June 9-13, 2007, San Diego, California, USA. Copyright 2007 ACM 978-1-59593-706-3/07/0006 ...$5.00.

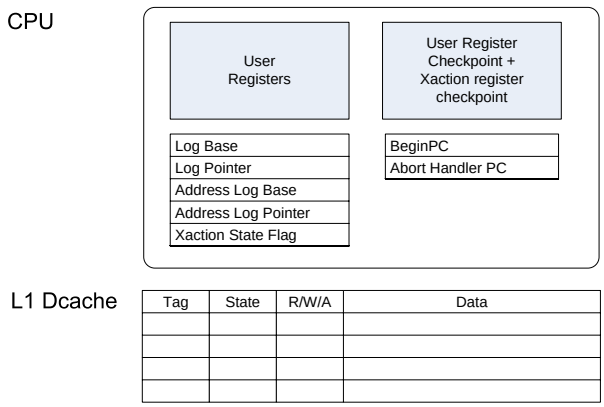

Figure 1: CPU Register Additions and Cache Line Additions

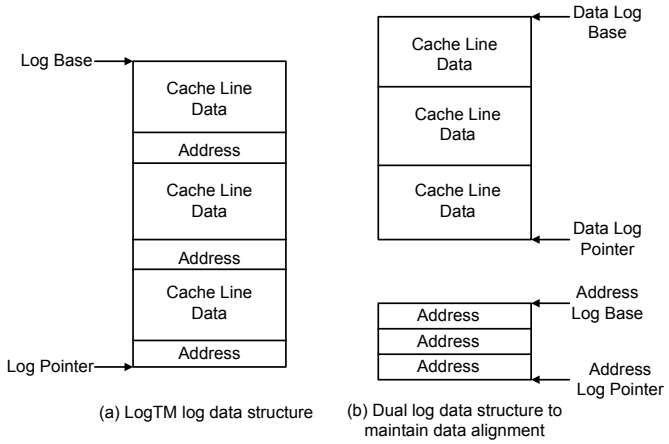

Figure 2: Original LogTM log structure and the dual log structure used in this implementation.

In this paper we deconstruct and duplicate the LogTM implementation of transactional memory in the M5 simulator. We deconstruct the OS requirements needed for a user level HTM implementation and quantify the interference the OS creates with transactional threads, leading to aborts. In our investigations we find the OS interference is highly benchmark dependent, with the OS causing little or no aborts in the majority of our benchmarks and up to 42% of aborts in one of our benchmarks. We add one new benchmark based on the JM H.264 decoder[21] using larger transactions which exposes pathological behavior in the current LogTM implementation. We find that over 98% of the transactions abort in this benchmark due to a very large read and write set size. The SPLASH2 benchmarks used in the original LogTM paper, however, exhibit 5% or less of transactions aborting.

The rest of this paper is organized as follows. Section 2 covers our implementation of LogTM in the M5 simulator and the necessary OS modifications we made to the Linux kernel. Section 3 gives an overview of the benchmarks we ran and analyzes the results we obtained. Section 4 briefly covers related work in the field and lastly, Section 5 covers future work we plan on pursuing and concludes the paper.

2. IMPLEMENTATION

Our implementation of LogTM consists of two parts. The first part is the hardware support implemented in M5 that provides conflict detection and logging. The second part is the OS support required for LogTM. This section describes in detail our version of LogTM including the similarities and differences from the original version[17].

2.1 Hardware Implementation

To model the hardware we use the M5 Full System Simulator[4] developed at the University of Michigan. M5 is unlike the GEMS[14] simulation environment used by the creators of LogTM in several ways. First, the GEMS simulation environment uses the Virtutech Simics ISA simulator to run code, and then uses a back end timing accurate memory simulator called Ruby. In this setup LogTM is visible only to Ruby, but not to Simics. On the other hand, M5 is a monolithic simulation environment, modeling the entire system including the OS and peripherals. This is a key difference because M5 can simulate all the interactions between the OS and hardware allowing us to quantify the effect the OS has on a TM system transparently, whereas the GEMS system cannot. Secondly the original LogTM was created for a SPARC ISA system, and our version is implemented on a Alpha ISA. This has implications regarding unaligned memory accesses that will be discussed later in section 2.2. Another key difference between GEMS and M5 is that M5 actively moves the data around its simulated memory hierarchy, allowing us to verify the coherence protocol and TM extensions are properly manipulating the memory state. Ruby in this case deals just with modeling the timing of memory accesses and Simics keeps its own internal memory state[14].

2.2 Processor additions

To duplicate LogTM we use many of the same hardware constructs as in LogTM[17]. *Figure* 1 shows the extra hardware needed in the CPU to implement our TM system. Our additions to the core, as was done in LogTM, include a register checkpoint that saves off all the user mode register state when a *begin_transaction* instruction is encountered. We also have the *BeginPC* register which stores the PC of the *begin_transaction* instruction, and the *AbortHandler* register that contains the address vector to the software handler for doing an abort.

To maintain the logs we use four registers, because, unlike LogTM, our version has two logs instead of one. Each pair of registers points to the base and end of their respective logs. The reason for two logs is a result of memory alignment issues in the simulated Alpha hardware. The Alpha has no hardware support for unaligned memory accesses. Therefore the 64-byte cache line in our simulated hardware and the 8-byte wide virtual address must be stored in different logs, because the Alpha hardware can not store a 72-byte block in contiguous memory without causing a trap to its unaligned access handler. Alternatively, we could have built the support in for writing 72 bytes contiguously, but decided against this for the simpler two log solution. *Figure* 2 shows how our log is organized in comparison to the original LogTM implementation. So when a write occurs in a transaction, the write is split into three memory operations,

the original store, a store of the old cache line to the data log, and a store of the virtual address of the saved line to the address log. We model these operations as taking place concurrently in the L1 cache, making our cache triple ported internally. This is identical to the original LogTM. An alternative is to break up the transactional write and make it take multiple cycles to complete. We chose not to pursue this alternative so as to duplicate LogTM as fully as possible. As in LogTM the whole cache line is stored in the data log due to the way the cache coherence mechanism works. The logs are stored in a private section of each thread's virtual memory, so in theory the logs can be as large as needed and are not dependent on cache size. Because the logs are in virtual memory, a transactional write can potentially lead to three separate page faults meaning our LogTM can incur extra OS overhead when in a transaction.

The *XactionState* register is similar to its analog in LogTM, but it has more than just two states, inside or outside of a transaction, as in LogTM. Our version has four states as follows: State 0: No transaction, State 1: In transaction, State 2: Aborting, State 4: Suspended transaction. The first two states are self explanatory, but the other two are necessary for subtle reasons pertaining to the operating system. Abort mode is used to detect and guard against recursively entering the abort routine and executing it multiple times. This is illustrated with the pseudo-code in *Example* 1. When a kernel handler begins, its first action is to call the abort routine in the case of exceptions, faults or interrupts. When a conflict is detected between transactions we issue a system call, calling the abort routine directly. When entering the abort routine, it checks to see if the CPU is already in abort mode executing the handler, and returns immediately if this is true. This check is needed because we can enter the routine multiple times due to page faults from loads or stores that are accessing user memory from the kernel. *Example* 1 shows we execute two loads and one store to user space to roll back the log. If either of these page fault, the kernel will trap to its page fault handler, which will attempt to run the abort routine again. Therefore the abort mode is necessary, either implemented in hardware or software, to prevent us from running the abort routine multiple times. When in the abort state reads and writes behave the same as when no transaction is active. While the abort mode could be set as a flag in part of the user address space, we kept it in the state register because all the other states need to be kept in the state register to modify behavior of load, stores, and the hwrei instruction as is described in the paragraph below. We viewed it as being simpler to encode all the states into the same state register, and not have it split up into different locations. Mode 4 is a special mode to take care of escape actions[18] in the system. Escape actions are special cases where a transaction is not aborted due to an OS event. In the Alpha architecture, PAL code takes care of many of the faults experienced by the CPU. TLB fills are an example where the CPU traps to PAL code but does not enter the OS. We do not want to abort for the cases where the CPU briefly enters PAL code and returns, so we allow the CPU to enter mode 4 upon entering PAL code, and then transition back to mode 1 upon returning to user code. With the Alpha hardware, we perform the transition from mode 4 back to mode 1 when executing a modified *hwrei* instruction. The modified instruction checks the *XactionState* register and the program counter value being jumped to, when in mode 4 and jumping back to a user code address we reset *XactionState* to 1. When in mode 4, reads and writes behave as if there is no transaction active.

Example 1 Pseudo-code for the abort routine and example kernel handler function.

```
kernel_handler() {
  abort_routine();
  ...
  handler();
  ...
  return;
}

abort_routine() {
  SET_NO_INTERRUPTS(true);
  READ_XACTIONSTATE(state);
  if (state == Abort)
    SET_NO_INTERRUPTS(false);
    return;
  else if (state == Suspended)
    MODIFY_RETURN_ADDRESS();

  SET_XACTIONSTATE(Abort);

  log_ptr = READ_LOGBASE();
  end_log = READ_LOGPTR();
  addr_ptr = READ_ADDRLOGBASE();
  end_addr = READ_ADDRLOGPTR();

  while (addr_ptr != end_addr)
    address = LOAD(addr_ptr);
    for (i = 0; i < sizeOfCacheBlk; i++)
      data = LOAD(log_ptr);
      STORE(data, address);
      address+8;
      log_ptr+8;
    addr_ptr+8;

  RESET_LOG_POINTERS();
  SET_XACTIONSTATE(NoXaction);
  SET_NO_INTERRUPTS(false);
  return;
}
```

2.3 Memory System additions

As seen in *Figure* 1, the cache also has some added hardware. Each cache line in the L1 cache has three additional state bits. These bits are R, W, and A. The R and W bits are from the original LogTM implementation while we have added the A bit to ours, being used in conjunction with the *XactionState* state register set equal to 2. The R bit signifies a transaction has read this line. The W bit signifies a transaction has speculatively written into the cache line. The A bit signifies a cache line with the W bit set is rolling back. On commit, the cache clears all the R, W, and A bits, setting them to false. The R bits are set to false and the A bits are set to true for any W bits set to true when the CPU detects the *XactionState* register transitioning from any state to the abort state. The A bit is used to allow the cache to release a cache line early during the abort routine.

Example 2 Pseudo-code for example PAL code routine.

```
PAL_routine() {
  ...
  if (CanHandleInPal);
    handleInPAL();
    return;
  else
    switch_protection(Kernel);
    kernelHandler();
    switch_protection(User);
    return;
}
```

Since the cache only sees loads and stores as either transactional or non-transactional as stated in section 2.2, the A bit is used. When the cache writes to the last word in the cache line, if the A bit is set, the cache will clear the W bit and A bit, releasing it early from the aborting read/write set. As can be seen in *Example* 1, the use of the A bit is implicit, when the inner loop is finished, the cache will automatically release the line. We decided to do this over the alternative, which is to use an instruction to force the cache to clear all its transaction state bits when executed. Due to time constraints we did not investigate if this early release mechanism offers any improvement.

For conflict detection, we use the coherence protocol just as in LogTM. We use the MOESI cache coherence protocol in our implementation as in the original LogTM. There are three types of memory operations the caches can snoop: regular memory accesses, regular memory accesses to another transaction's read/write set, and transactional memory accesses. The cases and responses are summarized in *Table* 1. For the case where a regular memory access is snooped by a remote cache to a non-transactional line, the normal MOESI mechanism is used. For the case when a regular memory access snoops a transactional line we *nack* the request and make it retry later in all cases where the remote cache would supply the data. The reason for *nacking* the access is to not allow the non-transactional part of the program to see any speculatively accessed and modified data. This method of *nacking* could lead to starvation in a pathological case where a transaction prevents another thread not in a transaction from making progress because the non-transactional thread is attempting to read from a transaction's read/write set, but we ignore this pathology for now. For the case when one transactional memory access hits to a line in another transaction, we have to determine if there is a conflict that would cause one of the transactions to roll back. If a transaction requests a read that hits another transaction's read set, then there is no conflict and both transactions will proceed. If a transaction requests a read that hits in another transaction's write set, there is a conflict and we need to cause a rollback in one of the transactions. Similarly a conflict is present if a write hits in another transaction's read set. The last case is when a write tries to write to a line in another's write set. To deal with conflict management, our current method is to force the transaction that generated the request to rollback if any other cache raises the conflict line. This contention management scheme can possibly lead to live lock. We believe smarter conflict arbiters can prevent live lock, but leave exploring this for future research.

Example 3 Pseudo-code for system call stub in user TM program.

```
abort_sys_call_stub() {
  SETXACTIONSTATE(NoXaction);
  syscall_trap(abort_routine);
  RESTORE_CHECKPOINT();
}
```

Currently, unlike LogTM, we do not support stalling conflicting transactions or allow transaction state spilling from the L1. The reason for not supporting spilling transactional state from the L1 is because we currently use a snoopy bus protocol and interconnect instead of a mesh network interconnect with a directory protocol. In LogTM, the directory protocol allowed for "Sticky" M bits[17] to indicate data had spilled from the L1 because the L2 contains the tag directory to map all the tags present in the cache hierarchy. With a snoopy protocol, we do not have "sticky bits". While we could have a implemented a spilled bit to indicate data had spilled, we would have no way to resolve what has spilled. This would potentially lead to one transaction with spilled data being allowed to lock out all the other transactions until it completed. The "sticky" M bits on the other hand allow for transactions to determine which lines were spilled from the L1. For future work we will be moving to signatures such as the ones in LogTM-SE[24] which greatly simplifies dealing with transactions that may spill from than the L1 cache. Not supporting spilling for this paper was not an issue though, as we sized our caches appropriately which is explained in section 3.1.

2.4 Linux OS Modifications

We found when duplicating LogTM in M5 we needed to make two major changes to the Linux operating system for the correct operation of our TM extensions. The largest modification we made was putting the transaction abort routine code into the kernel itself. We do this primarily because of the nature of the trap hardware in the Alpha. All events that can lead to entering the kernel first appear as escape actions. We are not able to determine if the hardware will actually enter the kernel until the code jumps to it because it first attempts to handle the trap in PAL mode as shown in the pseudo code in *Example* 2. Once in the kernel, we run the rollback routine to detect if we were in a transaction. We do this as shown in *Example* 1, by checking if the suspended state is set. If the suspended state is set, we modify the return address the kernel will return to. We set the return address to point at the RESTORE_CHECKPOINT() routine as shown in *Example* 3 at the end of our new system call stub function, from there the transaction will then be restarted. We need to rollback whenever we enter the kernel because we can not safely handle any kernel events in a transaction. Examples include preemptive context switches due to interrupts, and page faults, which is similar to the original LogTM. System calls are also kernel events, but we make sure to strictly avoid using system calls in a transaction. A benefit and second reason for putting the rollback code in the kernel is to let it run without interrupts enabled. This allows us to guarantee the rollback completes, because otherwise the system could take an interrupt and leave lines in the cache indicating they are still part of a

Remote Requester	Request	Local State Bits R W A	Conflict?	Response to Requester
Non-Transaction	Read	T F F	Yes	Nack
Non-Transaction	Read	X T X	Yes	Nack
Non-Transaction	Write	T F X	Yes	Nack
Non-Transaction	Write	X T X	Yes	Nack
Transaction	Read	T F F	No	None
Transaction	Read	X T X	Yes	Abort
Transaction	Write	T X X	Yes	Abort
Transaction	Write	X T X	Yes	Abort
Transaction or Non-Transaction	Read/ Write	F F F	No	None

Table 1: Snoop responses to transactional and non-transactional memory operations. (T = True, F = False, X = Don't care)

transaction. This can cause other transactions to conflict and abort, leading to a convoying effect of all the conflicting transactional threads. These threads would then spin waiting for the aborting thread to be swapped back in and complete its abort before the others can continue. This convoying effect is something TM is supposed to avoid, which is a strong argument for placing the abort routine in the kernel as well. It is possible to have the abort routine in user space for the common case of aborts, but the kernel would still need to have semantics to complete an in progress abort if it interrupted a thread doing an abort. With the abort handler now in the kernel, we also add a new system call for user code to access the abort routine when the thread is forced to abort from outside the kernel.

The second modification needed to the kernel is just as important but not not as subtle as needing the abort handler in the kernel. The kernel also needs to be modified to save the transactional state to the kernel thread context data structure. If this is not done, a context switch from the application to another process will corrupt the transactional state or worse, potentially leading to multiple threads writing to the same rollback logs.

3. RESULTS

To test our implementation of LogTM we ran a mixture of benchmarks. We used two hand written micro-benchmarks, four SPLASH2[23] benchmarks converted to use transactions and one new benchmark which is a hand coded parallel version of the JM H.264/AVC decoder. All these benchmarks were run on the M5 simulator. Although we can not compare directly to the results of LogTM since our version of LogTM has a some differences, we do compare general trends and conclusions made by the original authors.

3.1 Simulator Setup

The simulation environment is set up as specified in *Table* 2. Each CPU is modeled as a single instruction, single cycle simple core running at 2GHz. The overall system is modeled as a CMP with a single shared L2, with separate L1s for each CPU. We use relatively large and highly associative L1 caches to prevent transactions in our particular benchmarks from spilling. Even though LogTM can handle spilling and we can not, this difference does not matter because the larger L1s can hold even our largest transactions. We did not see any evidence of the results being skewed by doing this. The L1 caches are connected by a shared bus running at 2GHz, the same as the CPU cores. The main memory is connected to the L2 and has a latency of 100 cycles.

3.2 Benchmarks Overview

The benchmarks we use are drawn from the pool of benchmarks commonly used in current TM research. We use two micro-benchmarks that provide some insight into the overheads and pathologies inherent in TM in comparison to locks. We also use four SPLASH2 benchmarks modified to use transactions. Our last benchmark is a hand modified version of the H.264 decoder from the JM group[21]. A brief description of each benchmark is included below.

Shared Counter: This micro-benchmark creates ten threads that all compete to increment four global counters one million times each. This benchmark is designed to test how well transactions work when contending for data locations that are accessed constantly by many other threads.

Sorted Linked List Insert: This micro-benchmark creates ten threads that all attempt to insert into a sorted linked list about 500 hundred times each. Each thread randomly creates values for its nodes and then calls the insert function for the linked list object which protects the entire list with a transaction.

Number of cores	4
Frequency	2GHz
L1 DCache	Private 64kB, 8-way associative, 64-byte line size
L1 ICache	Private 64kB, 8-way associative, 64-byte line size
L1 Latency	1 cycle
Interconnect	Shared bus running at 2GHz
L2 Cache	Shared 2MB, 16-way associative, 64-byte line size
L2 Latency	10 cycles
Main Memory	1024MB
Main Memory Latency	100 cycles

Table 2: M5 Simulator Setup Parameters

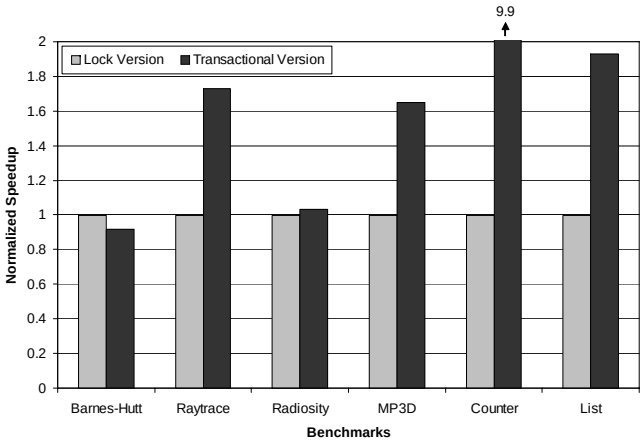

Figure 3: Normalized speedup of TM versions compared to lock versions of the benchmarks

Raytrace: This benchmark spawns multiple threads that then render a scene using the raytrace algorithm. For our tests we use the teapot model as the input into this benchmark.

Barnes-Hut: This benchmark simulates the gravitational attraction forces experienced between many bodies in a three dimensional space and computes each body's new velocity and position during each step of the simulation. The main shared structure in this benchmark is a multidimensional tree that divides up the search space the bodies are present in. We run this benchmark using a reduced input set of 512 bodies and four simulation steps just as in the original LogTM publication.

MP3D: This benchmark simulates a virtual wind tunnel. It models particles of air flowing over a CAD model. The main shared data structure is the 3D space of the wind tunnel divided up into cells, as the particles move the cells are updated with which particles they contain. This benchmark was originally part of the SPLASH benchmark suite. This benchmark is run using 4096 particles and 50 time steps.

Radiosity: This benchmark is like raytrace, it renders a scene by computing the way light interacts with the geometry in the scene. The main shared data structure is a binary space partitioning tree that holds the geometry, and the geometry itself by having the amount of light being emitted by it updated on each iteration of the algorithm. We use the built in room model when running this benchmark.

H.264 Decoder: This benchmark is still being actively developed, and only the deblocking filter has been parallelized. The deblocking filter does make up a large portion of the serial runtime[2] and it displays inherent parallelism so we concentrated on augmenting this part of the decoder first. We are including it because it is an example of a potential real life application that would benefit from using multiple processing cores. This benchmark decodes four frames of a H.264 compressed video stream.

3.3 Analysis

The results are presented in *Figure* 3 and *Table* 3. *Figure* 3 presents the normalized speedup of the benchmarks compared to their lock based versions. *Table* 3 breaks down the characteristics of the transactions for each benchmark. This includes the percentage of transactions that aborted, the percentage of aborts caused by conflicts with other transactions, percentage of aborts caused by the OS and the average read and write set size measured in bytes.

The results we obtained for the SPLASH2 benchmarks exhibits the same trends and general speedup as in the original LogTM paper. Raytrace has a very low conflict rate and very small transactions as can be seen in *Table* 3, and it obtains a sizable speed up over its original lock based code as seen in *Figure* 3. Radiosity has a relatively high conflict rate, but shows a slight speedup, which is different from the original LogTM paper. The original LogTM had a very low percentage of conflicts. This may be due to the fact we do not support stalling, and could be leading to higher abort rates, we may also be seeing this behavior due to the fact we simulate with 4 cores instead of 32. MP3D shows a significant speedup over its locking counterpart, but it has essentially no conflicts at all. Suggesting that the performance gains may be due to the fact that only one instruction is needed to begin a transaction instead of many to obtain a lock. Barnes exhibits the same trends with a small abort rate, just like raytrace and MP3D. It compares favorably to its lock based code, performing within 10%. The slow down experienced can in part be due to the conflicts experienced between transactions and the OS forcing more aborts. It is interesting to note that Barnes has 42% of aborts being caused by the OS. During our investigation we discovered that Barnes causes a high degree of page faults, which in turn cause the high percentage of aborts. We believe these page faults are being caused from the way Barnes is coded. Barnes also rebuilds its shared tree data structure on every iteration of the algorithm. The tree occupies a large portion of memory, and the building of the tree is done using transactions. Since the bodies move around inside the 3D space represented by the tree, on every iteration of the algorithm each processor can be touching a different part of memory when inserting its bodies into the tree. We believe this is the main cause of the page faults and why we are seeing a high percentage of the aborts being caused by the OS. This is in contrast to the other benchmarks seeing 1% or less of aborts being caused by the OS. Since we rollback every time we enter the kernel as described in section 2.4, we repeat more computation than we would otherwise with just locks.

For the micro-benchmarks, we get interesting results. The counter micro-benchmark, even though it should have a high conflict rate, does not. The OS in this case also has no effect on the aborts experienced. This is due to the fact that the transactions are small and complete quickly. For the list insert benchmark we see that it has a rather large 25% conflict rate. Even with this large conflict rate, it still performs better than its locking version primarily due to the extremely low overhead transactions incur compared to pthread mutexes. Pthread mutexes are rather expensive on the Alpha hardware, requiring multiple load locked and store conditional instructions to grab the lock. Still it is interesting to note that the transactions abort often, and this means that the list insert benchmark is experiencing pathological behavior in LogTM. This is due to the current contention management technique, and the very large read set size. When the linked list grows large, any thread attempting to insert a node and write to part of the list has a high probability of conflicting with a another transaction. A solution to

Benchmark	Percent of Transactions Aborted	Percent of Aborts that Conflicted	Percent of Aborts caused by OS	Avg Write Set Size in Bytes	Avg Read Set Size in Bytes
Barnes-Hut	2.0	57.8	42.2	378.1	455.5
Raytrace	0.7	99.7	0.3	127.0	395.4
Radiosity	36.1	99.9	0.1	103.9	213.1
MP3D	0.4	98.1	1.9	180.3	258.6
Counter	6.7	100.0	0.0	256.0	256.0
List Insert	25.5	98.8	1.2	191.6	11575.3
JM H264/AVC	98.8	99.9	0.1	1795.3	5758.1

Table 3: Breakdown of transaction characteristics of the tested benchmarks into percentage of transaction aborts, percentage of aborts caused by other transactions, percentage of aborts caused by OS interference, average write set size in bytes and average read set size in bytes.

this is either to allow the programmer to release part of his read/write set early, knowing it will not be touched again, which is similar to open nesting[18], or devise a smarter contention manager. Which solution will ultimately be better needs to be investigated in future work.

The last data point in *Table* 3 is for the parallel version of the H.264 decoder. We threaded the deblocking filter by breaking up the main loop that filters each block of pixels into separate threads. We put transactions around the sections of the loop body that worked on the block of pixels assigned to each iteration. For future work we will pursue a more aggressive parallelization of the H.264 decoder by threading other parts of the decoder. We will also consider alternatives such as functional parallelization instead of doing a speculative loop parallelization as we do here. It has attributes that are very different from the SPLASH2 benchmarks presented and the database benchmarks used in past literature. It has a very large average read and write set size, as can be seen, about an order of magnitude larger than any of our other workloads, except for list which has an even larger read set. This means it is not an entirely accurate assumption that transactions are going to be small in future applications. The benchmark also incurs a very large abort rate, primarily because the transactions are long running as evidenced by the large read and write set and the current method to decide which transaction to abort. This again implies the need for a much smarter contention manager to break out of this pathological case being seen here. Because of this pathological performance, no runtime data is presented because the simulations never finish due to them being live locked. We were able to gather runtime results for the lock based version. By threading the deblocking filter, we are able to increase the performance over the original single threaded code by 20%. We believe TM can also attain this performance by using a smarter contention manager. Even though this benchmark is just a single data point, we believe this is a truer representation of future applications that will use TM to extract performance from multi-core processors. We believe this to be true because future parallel programs will most likely protect data conservatively, using larger transactions. Also, currently sequential programs being ported to work with multi-cores will not be as likely to have the optimized parallel data structures and algorithms as found in current scientific and database benchmarks.

4. RELATED WORK

There has been substantial work in the field of transactional memory lately that runs the full spectrum, from purely software implementations to hardware implementations and versions combining both. The first style of TM was proposed by Herihly and Moss[13], this is a hardware technique using the cache coherence policy to facilitate conflict detection while using a side cache to buffer speculative writes. Transactional Consistency and Coherence (TCC)[10] was proposed by Stanford and it is very similar to Moss' version of TM. TCC puts speculative writes in a side buffer but, it differs by getting rid of cache coherence and instead uses a two stage commit arbitration protocol to decide which thread gets to commit its transaction. Contrary to TCC, Wisconsin has developed LogTM and offers a different view on TM. It uses cache coherence for conflict detection like Moss, but unlike Moss, it eagerly writes values into main memory and saves old values to the side. LogTM assumes aborts happen rarely. This version of TM uses a software handler to process aborts, and needs less hardware than TCC. Similarly to LogTM, there are other types of TM that use the coherence scheme to detect conflicts. UTM[3] is similar to LogTM in that it uses eager conflict detection and eager version management. VTM[19] uses lazy version management instead, but still uses the cache coherence protocol to do eager conflict detection.

Software TM (STM) research has also been very active with many implementations being proposed. This type of TM is very attractive because it requires no hardware support, but it is usually very slow due to the requirement for a very heavy handed runtime to detect conflicts[12][20]. STMs also suffer from not being able to compose arbitrary critical sections very well, because of the need to tie the transactions to a specific data structure or object. One STM[11] does offer a solution to these problems and performs rather well, but it needs to instrument every read and write in the program leading to high overhead.

Other proposals exist that attempt to combine the best of both STM and HTM into hybrid systems [16][5]. These systems use HTM techniques until the transactions get too large or call system code and then gracefully fall back to a software technique for cases hardware can not handle. The biggest challenge for hybrid systems is maintaining isolation between software only and hardware only transactions efficiently in these systems.

5. CONCLUSION AND FUTURE WORK

In this paper we have duplicated the LogTM HTM in the M5 simulator and investigated the similarities and differences between our implementation and the original. We also have detailed the OS support needed and have discovered it involves extending parts of the kernel. From the results we have seen that TM still has many areas to improve in with respect to some benchmarks, especially the H.264 benchmark. This benchmark in particular is contrary to the assumptions made by previous TM papers about the nature and size of transactions, with the read and write set in the thousands of bytes. It is also contrary to the assumption the percentage of conflicts is small, instead 98% of the transactions conflict. This points to the need to develop more parallel benchmarks to determine what should be the next step to make TM truly viable for unlocking the power of future multi-core chips. Another surprising benchmark is the sorted linked list insert benchmark that exhibits a very high conflict rate of 25%. It is surprising to see this simple benchmark exhibiting such pathological behavior. Other than these two benchmarks, the others track the trends shown in LogTM with the transactions being small and having a very low conflict rate of approximately 5% or less, with the exception of Radiosity. We have also quantified the interference an OS can cause in a TM system. We find that the OS very rarely causes transactions to abort due to interrupts or faults. About 1% of aborts were caused by the OS. However, with Barnes from SPLASH2, 42% of aborts were caused by the OS showing that the OS can have a significant impact on aborts. These results have exposed some of the problems inherent in HTMs and pointed to some of the future directions HTMs needs to pursue, such as more representative benchmarks and smarter conflict handlers.

In future work we plan on investigating further into operating system support for TM and also on better conflict resolution schemes to prevent TM from entering some of the pathological cases we have seen exhibited in our benchmarks. We also want to work further on developing benchmarks that will be representative of future applications that will take advantage of multiple cores that are not scientific applications like the SPLASH2 suite, or database applications. Such future applications will likely be concentrated in multimedia applications in particular, similar to our H.264 benchmark.

6. ACKNOWLEDGEMENTS

We would like to thank our shepherd Trey Cain, as well as the anonymous reviewers for their feedback on the drafts of this paper. This work has been supported by ARM Ltd.

7. REFERENCES

[1] International technology roadmap for semiconductors: System drivers, 2006.

[2] M. Alvarez, E. Salami, A. Ramirez, and M. Valero. A Performance Characterization of High Definition Digital Video Decoding Using H.264/AVC. In *IEEE International Symposium on Workload Characterization*, 2005.

[3] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 316–327, Washington, DC, USA, 2005. IEEE Computer Society.

[4] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The m5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006.

[5] Colin Blundell, E Christopher Lewis, and Milo M. K. Martin. Unrestricted transactional memory: Supporting i/o and system calls within transactions. Technical Report CIS-06-09, Department of Computer and Information Science, University of Pennsylvania, Apr 2006.

[6] Luis Ceze, Pablo Montesinos, Christoph von Praun, and Josep Torrellas. Colorama: Architectural support for data-centric synchronization. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*. Feb 2007.

[7] Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 227–238, Washington, DC, USA, 2006. IEEE Computer Society.

[8] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 338–349, New York, NY, USA, 2003. ACM Press.

[9] Lance Hammond, Brian D. Carlstrom, Vicky Wong, Mike Chen, Christos Kozyrakis, and Kunle Olukotun. Transactional coherence and consistency: Simplifying parallel hardware and software. *IEEE Micro*, 24(6), Nov-Dec 2004.

[10] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 102, Washington, DC, USA, 2004. IEEE Computer Society.

[11] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402. Oct 2003.

[12] Maurice Herlihy, Victor Luchangco, Mark Moir, and III William N. Scherer. Software transactional memory for dynamic-sized data structures. pages 92–101, Jul 2003.

[13] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, New York, NY, USA, 1993. ACM Press.

[14] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, 2005.

[15] Austen McDonald, JaeWoong Chung, D. Carlstrom Brian, Chi Cao Minh, Hassan Chafi, Christos Kozyrakis, and Kunle Olukotun. Architectural semantics for practical transactional memory. pages 53–65. Jun 2006.

[16] Mark Moir. Hybrid transactional memory, Jul 2005. Unpublished manuscript.

[17] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. Logtm: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pages 254–265. Feb 2006.

[18] Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood. Supporting nested transactional memory in logtm. *SIGARCH Comput. Archit. News*, 34(5):359–370, 2006.

[19] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 494–505. IEEE Computer Society, Jun 2005.

[20] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 204–213. Aug 1995.

[21] Karsten Suhring. H264/avc reference software, 2007.

[22] E. B. van der Tol, E. G. Jaspers, and R. H. Gelderblom. Mapping of H.264 decoding on a multiprocessor architecture. In B. Vasudev, T. R. Hsing, A. G. Tescher, and T. Ebrahimi, editors, *Image and Video Communications and Processing 2003. Edited by Vasudev, Bhaskaran; Hsing, T. Russell; Tescher, Andrew G.; Ebrahimi, Touradj. Proceedings of the SPIE, Volume 5022, pp. 707-718 (2003).*, volume 5022 of *Presented at the Society of Photo-Optical Instrumentation Engineers (SPIE) Conference*, pages 707–718, May 2003.

[23] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: characterization and methodological considerations. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 24–36, New York, NY, USA, 1995. ACM Press.

[24] Luke Yen, Jayaram Bobba, Michael M. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. Logtm-se: Decoupling hardware transactional memory from caches. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture(HPCA)*. Feb 2007.